Software Tools for the Supervisory Control of Petri Nets Based on Place Invariants

Technical Report of the ISIS Group
at the University of Notre Dame
ISIS-2002-003
April, 2002

Marian V. Iordache and Panos J. Antsaklis
Department of Electrical Engineering
University of Notre Dame
Notre Dame, IN 46556, USA
E-mail: iordache.1@nd.edu, antsaklis.1@nd.edu

Interdisciplinary Studies of Intelligent Systems

Software Tools for the Supervisory Control of Petri Nets Based on Place Invariants*

Marian V. Iordache and Panos J. Antsaklis
Department of Electrical Engineering,
University of Notre Dame, IN 46556
E-mail: iordache.1, antsaklis.1@nd.edu

This document describes a Matlab toolbox for the supervisory control of Petri nets based on place invariants [8, 10]. In what follows we present a detailed description of a number of functions we propose. These are the topics we address:

- Solving integer programs
- Transformations to admissible marking constraints
- Enforcing linear constraints in Petri nets which may have uncontrollable and/or unobservable transitions
- Invariant computation
- The coverability (reachability) graph
- Deadlock prevention and liveness enforcement

This document does not present all functions of the toolbox, as some of the functions are typically used only as subroutines of the main functions. However, all functions are provided with detailed help lines which can be read using the `help` command of Matlab.

The software was developed under Matlab 5.3 and Matlab 6 on a SunOS platform.

*The partial support of the Lockheed Martin Corporation, the Army Research Office (DAAG55-98-1-0199), the National Science Foundation (NSF ECS99-12458), and the DARPA/ITO-NEST Program (AF-F30602-01-2-0526) is gratefully acknowledged.

M.V. Iordache, P.J. Antsaklis, “Software Tools for the Supervisory Control of Petri Nets Based on Place Invariants,” ISIS Technical Report ISIS-2002-003, April 2002.

Contents

M.V. Iordache, P.J. Antsaklis, “Software Tools for the Supervisory Control of Petri Nets Based on Place Invariants,” ISIS Technical Report ISIS-2002-003, April 2002.

IP_SOLVE - Interface to the Mixed Integer Program Solver LP_SOLVE

LP_SOLVE is a mixed integer program solver developed at the University of Eindhoven. Matlab does not have an integer program solver, so we selected LP_SOLVE for this purpose. Since LP_SOLVE is written in C, an interface had to be written in order to run it in Matlab. The procedure for the installation of the toolbox compiles LP_SOLVE together with the interface files, and so generates the Matlab executable file IPSLV, which contains the code of LP_SOLVE. The IP_SOLVE function translates its input into the input format of IPSLV, and then calls IPSLV.

The general form in which IP_SOLVE can be called is:

```
[res, how] = ip_solve(L, B, f, intlist, ub, lb, ctype)
```

In the format

```
[res, how] = ip_solve(L, B)
```

the function checks whether

$$Lx \geq B \text{ for } x \geq 0 \tag{1}$$

admits any integer solution. If it does, `res` is set to a feasible solution and `how` to `'ok'`. Else, `res` is set to an empty vector and `how` to `'infeasible'`.

In the format

```
[res, how] = ip_solve(L, B, f)
```

the function solves $\min f^T x$ subject to $Lx \geq B$, $x \geq 0$, and x integer. The function returns in `res` the optimal solution if the problem is feasible and bounded, a feasible solution if the problem is unbounded, and an empty vector if the problem is infeasible. The variable `how` is set to the type of problem: `'ok'`, `'unbounded'`, or `'infeasible'`.

In the general format `intlist` can be used to specify the entries of x which are to be integers. (Default is that all entries of x are to be integers.) Thus `intlist(i)`$\neq 0$ means that $x(i)$ is an integer.

Upper and lower bounds can be specified via ub and lb. When they are given, the condition $x \geq 0$ is replaced by `ub`$\geq x \geq$`lb`. By default `lb`$= 0$ and `ub`$= \infty$.

The argument `ctype` can be used to specify the direction of each inequality given by L and B. This is done as follows:

```
  L(i,:)x <= B(i) for ctype(i) = -1
  L(i,:)x  = B(i) for ctype(i) =  0
  L(i,:)x >= B(i) for ctype(i) =  1
```

All arguments except L and B are optional. If an optional argument is set to `[]`, IP_SOLVE will set it to the default value. In particular, when `f = []` the feasability problem is solved.

GETPN - Initialize Petri Net Object

GETPN generates a Matlab object of the *structure* class, representing a Petri net. The synopsis is:

```
[o] = getpn;
[o] = getpn(D);
[o] = getpn(Dm,Dp);
 . . .
[o] = getpn(Dm,Dp,Tuc,Tuo,m0)
```

The input is as follows

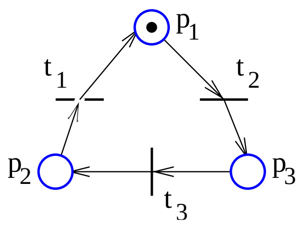
- `m0` is the initial marking.
- `D` is the incidence matrix, and `Dm` and `Dp` are the input and output matrices D^- and D^+ (i.e. `D = Dp - Dm`).
- `Tuc` and `Tuo` list the indices of uncontrollable and unobservable transitions, respectively. Note that i is the index of a transition t if and only if t corresponds to the i-th column of the incidence matrix.

The output is the Petri net object `o` such that

- `o.m0 = m0` is the initial marking (default is `o.m0 = []`).
- `o.Dm` and `o.Dp` are the input and output matrices (default values are `[]`).
- `o.Tuc = Tuc` and `o.Tuo = Tuo` list the indices of the uncontrollable and unobservable transitions, respectively. Default is `o.Tuc = []` and `o.Tuo = []`.

Example: In the Petri net below, all transitions are observable and the transitions t_1 and t_2 are uncontrollable. The code below creates the corresponding Petri net object.

```
m0 = [1 0 0];
Dm = [0 1 0; 1 0 0; 0 0 1];
Dp = [1 0 0; 0 0 1; 0 1 0];
pn = getpn(Dm, Dp, [1,2], [], m0);
```



M.V. Iordache, P.J. Antsaklis, “Software Tools for the Supervisory Control of Petri Nets Based on Place Invariants,” ISIS Technical Report ISIS-2002-003, April 2002.

PNCGRAPH - The Coverability Graph

This function computes the coverability graph (the usual construction; see [6, p. 171] or [9, pp. 66-71]). Note that for bounded Petri nets the coverability graph is identical to the reachability graph. The synopsis is

```
[Graph] = pncgraph(pnobj)
```

where `pnobj` is a Petri net object generated with GETPN. `Graph` is a cell array as follows:

- `Graph{1}{1} = m0`, the initial marking.
- `Graph{i}{1}` is the marking label of the node `i`.
- `Graph{i}{2}` is the list of (direct) predecessor nodes:
  `Graph{i}{2}{j}.n` is the number of the `j`th predecessor node;
  `Graph{i}{2}{j}.t` is the (index of the) transition by which node `i` is reached;
  in particular, if `Graph{1}` has no predecessors, `Graph{1}{2} = {}`.
- `Graph{i}{3}` is the list of (direct) successor nodes:
  `Graph{i}{3}{j}.n` is the number of the `j`th successor node;
  `Graph{i}{3}{j}.t` is the (index of the) transition by which the `j`th successor node is reached.

The object `Graph` can be visualized in text format using the function DISGRAPH. The function DISGRAPH is called by PNCGRAPH when run without an output argument. For instance, to see in text format the coverability graph of pn, type

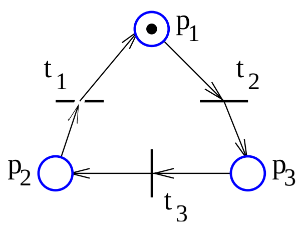
```
[Graph] = pncgraph(pnobj);
disgraph(Graph);
```

or

```
pncgraph(pnobj);
```

Example: The code below computes the reachability graph of the Petri net shown in the figure.

```
m0 = [1 0 0];
Dm = [0 1 0; 1 0 0; 0 0 1];
Dp = [1 0 0; 0 0 1; 0 1 0];
pn = getpn(Dm, Dp, [], [], m0);
pncgraph(pn);
```



The reachability graph is displayed as:

```
NODE 1   [1, 0, 0]
 IN: 3, t1;
 OUT: 2, t2;
NODE 2   [0, 0, 1]
 IN: 1, t2;
 OUT: 3, t3;
NODE 3   [0, 1, 0]
 IN: 2, t3;
 OUT: 1, t1;
```

The explain the text output, let's consider NODE 1. The first text line at NODE 1 indicates that NODE 1 corresponds to the marking $[1, 0, 0]$. The second and third lines indicate that NODE 1 has as input NODE 3 via firing t_3, and as output NODE 2 via t_2.

For Petri nets with enough many places, the markings are represented trough a list of places with nonzero marking. For instance, the marking $[0, 1, 0, 0, 3]$ would be represented as `p2:1, p5:3;`, signifying that the marking of p_2 is 1, the marking of p_5 is 3, and the marking of the other places is zero.

MRO_ADM - Transformation of Marking Constraints to Admissible Constraints using Matrix Row Operations

Description: Given the set of marking inequalities $L\mu \leq b$ and a Petri net with uncontrollable and unobservable transitions, the function attempts to transform the constraints to an admissible form $L_a\mu \leq b_a$. The new inequalities are related to the original inequalities by

$$\begin{aligned} L_a &= R_1 + R_2 L && (2) \\ b_a &= (R_2 + \mathbf{1})b - \mathbf{1} && (3) \end{aligned}$$

where all elements of R_1 are nonnegative, and R_2 is both diagonal and positive definite.

The function uses the matrix row operation algorithm of [8], pp.53-58, with two modifications. First, the matrix M is added a new column containing the initial marking:

$$M = \left[\begin{array}{cccc} D_{uc} & D_{uo} & \mu & \\ LD_{uc} & LD_{uo} & L\mu - b - \mathbf{1} & I \end{array} \right] \quad (4)$$

This allows us to declare failure as soon as one of the elements in the $L\mu - b - \mathbf{1}$ column becomes nonnegative during the matrix row operations. Second, the elements in the $M(n+1 \ldots n+n_c, .)$ part of the matrix are canceled by multiplication to the least common multiple, instead of using column zeroing (which is Algorithm 4 in [8]). This modification insures that the matrix R_2 is diagonal.

Synopsis:

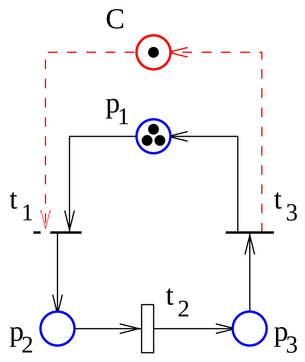
```
[La, ba, R1, R2, how, dhow] = mro_adm(L, b, D, Tuc, Tuo, m0, vrb)
```

where

- `D` is the incidence matrix
- `Tuc` enumerates the uncontrollable transitions. For instance, if `Tuc = [2 3]`, then the uncontrollable transitions are t_2 and t_3, and so the uncontrollable part of D is D_{uc} =`D(:,Tuc)`. The parameter is optional; the default value is `Tuc = []`.
- `Tuo` enumerates the unobservable transitions. The parameter is optional; the default value is `Tuo = []`.
- `m0` is the initial marking. Use `m0 = []` when the initial marking is not of interest. The parameter is optional; the default value is `m0 = []`.
- If `vrb` is zero, MRO_ADM has a silent execution. Else, messages may be printed. Default is `vrb = 1`.

- `how` is set to one of `'ok'` (all constraints have been successfully transformed) or `'not solved'` (one or more constraints have not been successfully transformed).
- For each constraint $L(i, .)x \geq b(i)$, `dhow{i}` is `'ok'` if this constraint has been successfully transformed, and `'not solved'` otherwise.

Example: The Petri net shown in the figure has all transitions observable and one transition uncontrollable: t_2. MRO_ADM is used to transform the inadmissible constraint $\mu_3 \leq 1$ to an admissible constraint. The code is



```
D = [-1 0 1; 1 -1 0; 0 1 -1];
m0 = [3; 0; 0];
Tuc = [2];
Tuo = [];
L = [0 0 1];
b = 1;
[La, ba, R1, R2, how, dhow] = mro_adm(L, b, D, Tuc, Tuo, m0);
```

The generated admissible constraint is $\mu_2 + \mu_3 \leq 1$. The control place enforcing it is the place C connected with dashed lines. The connections of the place C can be obtained from the input and output matrices of the closed-loop, `Dfm` and `Dfp`:

```
[Dfm, Dfp] = linenf(Dm, Dp, L, b)
```

ILP_ADM - Transformation of Marking Constraints to Admissible Constraints using Integer Programming.

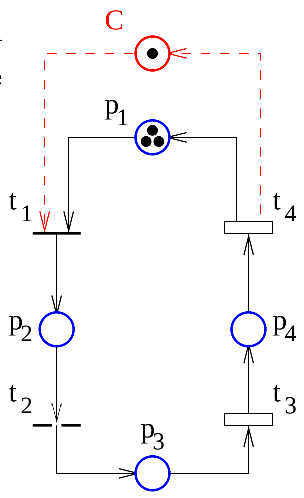
ILP_ADM has the same purpose as MRO_ADM, except that it uses integer programming instead of matrix row operations. This is a more powerful procedure than MRO_ADM, however it can be more computationally complex. Thus ILP_ADM is to be used when MRO_ADM fails.

The synopsis is:

```
[La, ba, R1, R2, how, dhow] = ilp_adm(L, b, D, Tuc, Tuo, m0, vrb)
```

The synopsis is identical to that of MRO_ADM, and so are the meanings of the parameters. So refer to MRO_ADM. The only difference is that `how` may also take the value `'impossible'`, when ILP_ADM detects that the transformation of (2) and (3) is impossible for one of the constraints of $L\mu \geq b$. The same is true of `dhow{i}`, which is set to `'impossible'` when ILP_ADM detects that the transformation of (2) and (3) is impossible for the constraint $L(i, .)x \geq b(i)$.

Example: The Petri net shown in the figure has all transitions observable and two transition uncontrollable: t_3 and t_4. ILP_ADM is used to transform the inadmissible constraint $\mu_2 + \mu_4 \leq 1$ to an admissible constraint. The code is



```
D = [-1 0 0 1; 1 -1 0 0; 0 1 -1 0; 0 0 1 -1];
m0 = [3; 0; 0; 0];
Tuc = [3, 4];
Tuo = [];
L = [0, 1, 0, 1];
b = 1;
[La, ba, R1, R2, how, dhow] = ilp_adm(L, b, D, Tuc, Tuo, m0);
```

The generated admissible constraint is $\mu_2 + \mu_3 + \mu_4 \leq 1$. The control place enforcing it is the place C connected with dashed lines. The connections of the place C can be obtained from the input and output matrices of the closed-loop, `Dfm` and `Dfp`:

```
[Dfm, Dfp] = linenf(Dm, Dp, L, b)
```

M.V. Iordache, P.J. Antsaklis, “Software Tools for the Supervisory Control of Petri Nets Based on Place Invariants,” ISIS Technical Report ISIS-2002-003, April 2002.

LINENF - Linear Constraint Enforcement in Petri Nets

Description: LINENF enforces linear constraints of the form

$$L\mu + Fq + Cv \leq b \tag{5}$$

where μ is the marking, q the firing vector, and v the Parikh vector (for all i, v_i is the number of firings of t_i since the initialization of the system). The use of the constraints involving μ and q has been described in [8]. The constraints $Cv \leq b$ can be used to describe fairness constraints (e.g. $v_3 - v_1 \leq 0$ specifies that t_1 should fire at least as often as t_3). When the PN has uncontrollable and/or unobservable transitions, the function uses PN transformations and MRO_ADM and ILP_ADM to transform the constraints to admissible constraints.

The function operates as follows. In the case of PNs with no uncontrollable and unobservable transitions, the input and output matrices of the supervisor enforcing (5) are:

$$D_c^- = \max(0, LD_p + C, F) \tag{6}$$

$$D_c^+ = \max(0, F - \max(0, LD_p + C)) - \min(0, LD_p + C) \tag{7}$$

where $\max(A, B, C, \ldots)$ is the matrix X of elements $x_{i,j} = \max(a_{i,j}, b_{i,j}, c_{i,j}, \ldots)$, $\min(A, B, C, \ldots)$ is similarly defined, and D_p is the incidence matrix of the plant PN. (Note that enforcing $Cv \leq b$ only involves adding the rows of $-C$ as rows to the incidence matrix.)

In the case of PNs with uncontrollable and/or unobservable transitions, the constraints (5) are transformed to marking constraints. The Parikh vector constraints are transformed as follows. Given a constraint

$$l^T\mu + f^Tq + c^Tv \leq d \tag{8}$$

for each transition t_i with $v_i \neq 0$ a new place p_i'' is generated, such that $\bullet p_i'' = t_i$ and $p_i''\bullet = \emptyset$. Then each $c_i v_i$ with $c_i \neq 0$ is replaced by $c_i\mu(p_i'')$.

The firing vector constraints are transformed using the indirect realization (section 7.2.2. in [8]). In this approach, given a constraint

$$l^T\mu + f^Tq \leq d \tag{9}$$

for each transition with $f_i > 0$ a new place p_i' and a new *uncontrollable* transition t_i' are added. Then (9) is transformed into a marking inequality. The transformation employed by LINENF is different from the one suggested in [8]. Thus, let $g = l^T D_p^-$. If $f_i > 0$, the term $f_i q_i$ is replaced by $(f_i + g_i)\mu(p_i')$; else the term $f_i q_i$ is deleted. Adding the g_i term allows LINENF to be effective on coupled constraints.

LINENF does for each constraint of (5) the operations above. Such a transformed marking constraint is further transformed to be admissible. Then, it is enforced in the transformed Petri net.

By collapsing the controlled net back to the original Petri net, the connections of the control place in the original net are obtained. The theoretical background of the algorithm can be found in [2].

Synopsis:

```
[Dfm, Dfp, ms0] = linenf(pn, L, b, F, C)
[Dfm, Dfp, ms0, how, dhow] = linenf(Dm, Dp, L, b, m0, F, C, Tuc, Tuo)
```

where unnecessary arguments can be omitted or set to `[]`. The notation is as follows:

- `pn` is a Petri net object.
- `Dm` and `Dp` are the input and output matrices of the plant Petri net.
- `Dfm` and `Dfp` are the input and output matrices of the controlled net. The last rows are the rows of the control places.
- `m0` is the initial marking. Use `m0 = []` if the initial marking is not of interest.
- `ms0` is the initial marking of the controlled net. If `m0 = []`, `ms0` is set to an empty vector.
- `Tuc` and `Tuo` specify the set of uncontrollable and unobservable transitions, respectively. For instance, `Tuc = [2, 3]` and `Tuo = 3` specify that t_2, t_3 are uncontrollable and t_3 is unobservable, where t_i stands for the transition corresponding to the i-th column of the incidence matrix. Note that in the LINENF convention, a transition may be unobservable and yet controllable.
- `how` is `'ok'` if all constraints have been successfully enforced, `'not solved'` if it failed without detecting that the system of constraints cannot be enforced, and `'impossible'` if one of the constraints cannot be enforced.
- `dhow{i}` reports the status of each constraint. Thus `dhow{i}` is `'ok'` if the constraint i has been successfully enforced, `'impossible'` if LINENF detected that the constraint cannot be enforced, and `'not solved'` otherwise.

It may be the case that we want to use the initial marking as a parameter for the marking of the control places. Then the following format is useful:

```
[Dfm, Dfp, ms0, how, dhow, Lf, Cf, bf] = linenf(pn, L, b, F, C)
[Dfm, Dfp, ms0, how, dhow, Lf, Cf, bf] = linenf(Dm, Dp, L, b, m0, F, C, Tuc, Tuo)
```

In this format the initial marking of the control places can be obtained as `bf` $-$ `Lf*m0`. Additionally, `Cf` allows us to specify the “invariant” enforced by the control places:

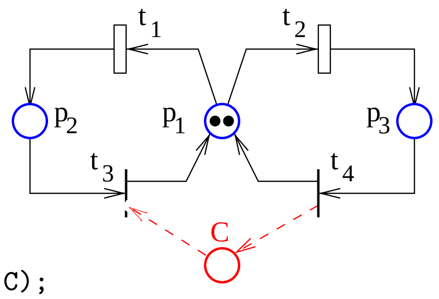
$$L_f\mu + C_f v + \mu_c = b_f \tag{10}$$

M.V. Iordache, P.J. Antsaklis, "Software Tools for the Supervisory Control of Petri Nets Based on Place Invariants," ISIS Technical Report ISIS-2002-003, April 2002.

where μ_c is the marking of the control places.

Example: The Petri net shown in the figure has all transitions controllable and two transition unobservable: t_1 and t_2. LINENF is used to transform the inadmissible constraint $v_1 - v_2 \leq 2$ to an admissible constraint, and to enforce the admissible constraint on the Petri net. The code is

```
D = [-1 -1 1 1; 1 0 -1 0; 0 1 0 -1];
m0 = [2 0 0]';
Tuc = []; Tuo = [];
L = []; F = [];
C = [1 -1 0 0]; b = 2;
[Dm, Dp] = d2dd(D);
pn = getpn(Dm, Dp, Tuc, Tuo, m0);
[Dfm,Dfp,ms0,how,dhow,Lf,Cf.bf] = linenf(pn, L, b, F, C);
```



The generated admissible constraint has the form $L_f\mu + C_f v \leq b_f$, and is $\mu_1 + 2\mu_3 + v_1 - v_2 \leq 2$. The control place enforcing it is the place C connected with dashed lines. The connections of the place C are obtained from the input and output matrices of the closed-loop, `Dfm` and `Dfp`.

Programs testing the linear constraint enforcement

A number of Matlab scripts with test data have also been generated. Most tests are taken from the examples in [8]. The tests have been put together in the script LINTST.

ISADM - Checks whether a set of constraints is admissible

Given a plant Petri net, it is important to answer the question of whether a set of desirable constraints on the operation of the net is admissible. A constraint is said to be *admissible* if its enforcement under the assumption that all transitions are controllable and observable does not attempt to inhibit uncontrollable transitions and does not detect unobservable transitions. The admissibility of a set of constraints is typically dependent on the initial marking. Therefore, when the initial marking is given, the function performs a reachability analysis to establish whether the constraints are admissible.

The synopsis is

```
[o] = isadm(pnobj, L, b)
[o] = isadm(pnobj, L, F, b)
[o] = isadm(pnobj, L, F, C, b)
```

where

- `pnobj` is a Petri net object (obtained with `GETPN`) describing the plant.
- `L`, `F`, `C` and `b` are the parameters of the constraints:

$$L\mu + Fq + Cv \leq b \tag{11}$$

- The output `o` is a vector evaluating the admissibility of each constraint of the set of constraints as follows:
  1. `o(i) = 1` if constraint i is admissible
  2. `o(i) = 0` if constraint i is inadmissible
  3. `o(i) = -1` if the admissibility of constraint i could not be decided.

For the reachability analysis, the function computes the coverability graph of the closed-loop Petri net. The closed-loop Petri net is the plant Petri net connected to the supervisor enforcing (11), obtained under the assumption that all transitions are controllable and observable. The function may not be able to decide admissibility only if the supervisor part of the closed-loop Petri net is unbounded.

Structural analysis with ISADM

When no initial marking is specified in `pnobj` or when `b = []`, the function performs a structural analysis to determine whether the constraint is admissible. In this case the output should be interpreted as follows:

1. if `o(i) = 1`, then the constraint i is admissible for all initial markings satisfying the constraints.

2. if `o(i) = 0`, then initial markings may exist such that constraint i is inadmissible.

The structural analysis implementation in ISADM is very fast, however, not as powerful as reachability analysis.

Example: Consider Petri net below modeling the three-tank probem from [8]. All transitions are observable, but only t_1, t_2, and t_3 are controllable. The following code can be used to verify that the constraint

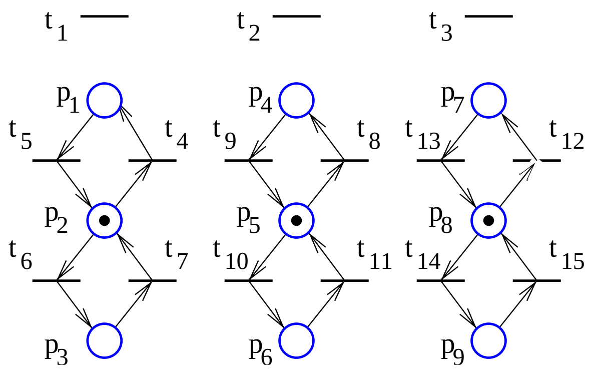
$$2q_1 + \mu_6 + \mu_9 - 2\mu_3 \leq 2$$

is admissible, and that

$$\mu_2 + \mu_5 + \mu_8 \geq 3$$

is inadmissible.

```
D1 = [ 1 -1  0  0; -1  1 -1  1; 0  0  1 -1;];
D = [zeros(9,3) [D1; zeros(6,4)] [zeros(3,4); D1; zeros(3,4)] [zeros(6,4); D1]];
Tuc = [4:15]; Tuo = [];
[Dm, Dp] = d2dd(D);
m0 = zeros(9,1); m0([2 5 8]) = 1;
pn = getpn(Dm, Dp, Tuc, Tuo, m0);
F = [2, zeros(1,14); zeros(1,15)];
L = zeros(2,9);
L(1,[3,6,9]) = [-2, 1, 1];
L(2,[2,5,8]) = -1;
b = [2; -3];
[v] = isadm(pn, L, F, b);
```



Note that structural analysis does not detect that the first constraint is admissible (at the given initial marking):

`[v] = isadm(pn, L, F, []);`

returns `v = [0, 0]`.

Test file

ISADMTST is a script containing examples for ISADM.

Invariant Computation

We may be interested to find the place or transition invariants of a Petri net. Matlab provides the function `null`, which finds a base of the null space of a matrix. For instance, if we desire to compute a base of the invariants satisfying $Ax = 0$ for some matrix A, the synopsis is

```
V = null(A,'r')
```

Then any invariant can be expressed as:

```
x = V*a
```

where `a` is a column vector. The second argument `'r'` insures that `V` is an integer matrix (when `A` is also).

Example: In the Petri net below, all place invariants are of the form αv, $\alpha \in \mathbb{R}$, for

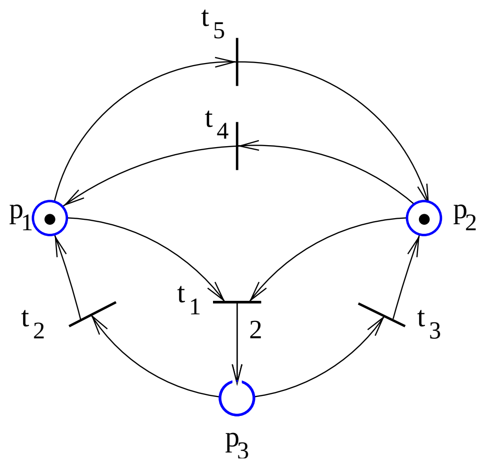
$$v = [\ 1 \quad 1 \quad 1\ ]$$

and all transition invariants are a linear combination (with real coefficients) of v_1, v_2, and v_3 below:

$$\begin{aligned} v_1 &= [\ 1 \quad 1 \quad 1 \quad 0 \quad 0\ ]^T \\ v_2 &= [\ -1 \quad -2 \quad 0 \quad 1 \quad 0\ ]^T \\ v_3 &= [\ 1 \quad 2 \quad 0 \quad 0 \quad 1\ ]^T \end{aligned}$$

The code below obtains this result:

```
D = [-1 1 0 1 -1; -1 0 1 -1 1; 2 -1 -1 0 0];

% Place invariant computation

v = null(D','r');

% Transition invariant computation

v = null(D, 'r');
```



Positive Invariant Computation

Problem Statement. We say that x is a positive invariant of M if $Mx = 0$, $x \geq 0$, and $x \neq 0$. Given a matrix M, we are looking to find a number of vectors v_1, v_2, … v_m such that

$$Mx = 0 \text{ and } x \geq 0 \Longleftrightarrow x = \sum \alpha_i v_i \text{ and } \alpha_i \geq 0 \tag{12}$$

The vectors v_1, v_2, … v_m satisfying (12) obviously satisfy: $v_i \geq 0$ for $i = 1 \dots m$. The **support** of a vector x is the set of indices j for which $x(j) \neq 0$; we denote by $\|x\|$ the support x. We consider invariants which are not equal to the null vector. For such invariants we say that x is a **minimal support invariant** if x is an invariant and there is no invariant y such that $\|y\| \subset \|x\|$.

It can be noticed that we can solve the problem of (12) if we take v_1, … v_m positive invariants with mimimal support, such that all supports of the minimal support positive invariants appear in $\|v_1\|$, … $\|v_m\|$, and $\|v_1\|$, … $\|v_m\|$ are distinct. Indeed, the $\Leftarrow$ direction of the proof is obvious, while the $\Rightarrow$ direction is as follows. To find the coefficients α_i, we first set them to 0. Next, note that a positive invariant x satisfies that there is v_i such that $\|v_i\| \subseteq \|x\|$. Then we set α_i to the largest real such that $x - \alpha_i v_i \geq 0$. Then let $x_2 = x - \alpha_i v_i$; if $x_2 \neq 0$, x_2 is a positive invariant, and so we apply to x_2 the same procedure as for x, in order to set a new coefficient α_i. Eventually we get to an x_n such that $x_n = 0$. At that time we know that $x = \sum \alpha_i v_i$.

Furthermore it can also be noticed that we have the minimum number of vectors v_1 … v_m solving (12) if we take them the minimal support positive invariants, as above.

Computational Complexity. The problem of finding the positive invariants may not be easy to solve. It is known that a matrix may have an exponential number of positive invariants with minimal (and distinct) support. For instance consider the matrix

$$N = [v_1,\, v_2,\, \dots v_n] \tag{13}$$

Assume that there is a single minimal support of positive invariants; so there is a positive invariant $\alpha \neq \mathbf{0}$ having that support, that is:

$$\alpha_1 v_1 + \alpha_2 v_2 + \dots \alpha_n v_n = 0 \tag{14}$$

and $\alpha_i \geq 0$, for all $i = 1 \dots n$. Then notice that the matrix

$$M = [v_1,\, v_2,\, \dots v_n,\, v_1,\, v_2,\, \dots v_n] \tag{15}$$

has 2^n distinct minimal supports of positive invariants! This proves that the number of minimal supports of positive invariants of a matrix with n columns can be larger than $2^{\lfloor n/2 \rfloor}$.

Implementation. The use of the function **INVAR** is proposed. This is a slight modification of INVAR1, which has been developed in our group in the past. The modification is that INVAR insures that the intermediary rows are integer. This solves the problems of INVAR1 which are known to the authors. INVAR1 is based on an algorithm by Martinez and Silva [7].

Synopsis:

```
C = invar(A)
```

The function computes a set of positive invariants with minimal support, and stores them as columns in `C`. Unlike INVAR1, invariants satisfying $Ax = 0$ instead of $x^T A = 0$ are computed.

Example: In the Petri net below, all positive place invariants are of the form αv, $\alpha \in \mathbb{R}_+$, for

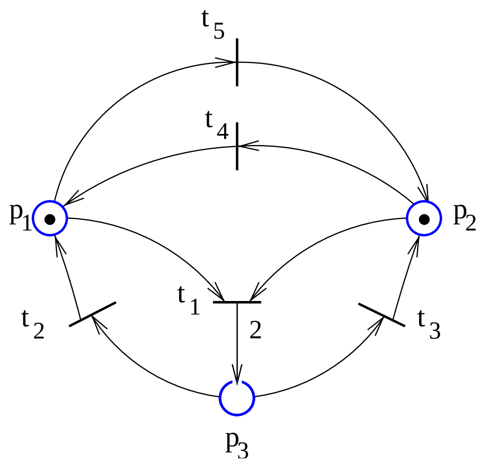
$$v = [\ 1 \quad 1 \quad 1\ ]$$

and all positive transition invariants are given by the $v_1 \ldots v_4$ below:

$$\begin{aligned} v_1 &= [\ 1 \quad 1 \quad 1 \quad 0 \quad 0\ ]^T \\ v_2 &= [\ 1 \quad 2 \quad 0 \quad 1 \quad 0\ ]^T \\ v_3 &= [\ 1 \quad 2 \quad 0 \quad 0 \quad 1\ ]^T \\ v_4 &= [\ 0 \quad 0 \quad 0 \quad 1 \quad 1\ ]^T \end{aligned}$$

The code below obtains this result:

```
D = [-1 1 0 1 -1; -1 0 1 -1 1; 2 -1 -1 0 0];

% Place invariant computation

v = invar(D');

% Transition invariant computation

v = invar(D);
```



DP - Deadlock Prevention and Liveness Enforcement

Description: This function can be used to generate deadlock prevention supervisors, liveness enforcing supervisors, and T-liveness enforcing supervisors.

> *Deadlock* is the state of a system that has become permanently blocked.
>
> *Deadlock Prevention* means to prevent reaching a state of total deadlock.
>
> *Liveness Enforcement* means to ensure that no local deadlocks occur. In the Petri net terminology, this means ensuring that all transitions are *live.*
>
> *T-liveness Enforcement* means to ensure that no deadlocks occur in the part of the system characterized by the set T. In the Petri net terminology, this means ensuring that all transitions in the set T are *live.*

DP generates by default deadlock prevention supervisors. They typically enforce liveness or T-liveness (for the largest possible set T). Options can be set to force *DP* generate supervisors guaranteed to enforce liveness or T-liveness, for arbitrary sets T. However, in the default mode *DP* is most likely to converge.

The supervisors are defined as follows. Given a Petri net, *DP* generates the marking constraints $L\mu \geq B$ and $L_0\mu \geq B_0$, such that when the Petri net is supervised according to $L\mu \geq B$, it is deadlock-free/live/T-live for all initial markings μ_0 satisfying $L\mu_0 \geq B$ and $L_0\mu_0 \geq B_0$. Furthermore, the constraints $L\mu \geq B$ are guaranteed to be admissible.

Synopsis: The following format can be used:

```
[L, B, L0, B0, how] = dp(pn, T, L, B, L0, B0, opt)
```

where the simpler formats below are also available:

```
[L, B, L0, B0, how] = dp(pn)
[L, B, L0, B0, how] = dp(pn, T)
[L, B, L0, B0, how] = dp(pn, T, L, B)
[L, B, L0, B0, how] = dp(pn, T, L, B, L0, B0)
```

Alternative formats which do not refer to Petri net objects are also available, and they are described in the help lines. The significance of the arguments is as follows:

- `pn` is a Petri net object created by GETPN. The initial marking (if any) of `pn` is ignored, as *DP* relies on structural net analysis.

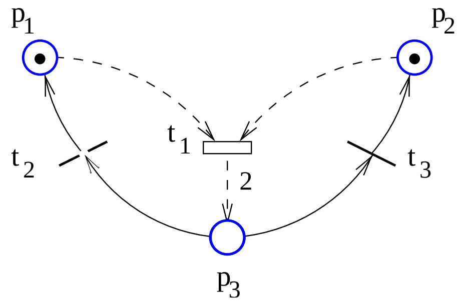
- `T` is the set of transitions T for T-liveness enforcement. To omit T, replace it with the empty vector `[]`. The default value of T is the set of transitions of the Petri net.
- `L` and `B` are the matrices L and B produced by the procedure. If their initial value is not empty, they can be given as input arguments. Nonempty initial L and B are interpreted as additional constraints $L\mu \geq B$ to be enforced on the Petri net. The inputs `L` and `B` can be omitted by replacing each of them with the empty vector `[]`.
- `L0` and `B0` are the matrices L_0 and B_0 produced by the procedure. If their initial value is not empty, they can be given as input arguments. Nonempty initial L_0 and B_0 are interpreted as constraints that all reachable markings satisfy, if the initial marking satisfies them. Initial L_0 and B_0 can be used to help the procedure converge. The inputs `L0` and `B` can be omitted by replacing each of them with the empty vector `[]`.
- `opt` is the options input. The options are of the form $opt = 8l + 4k + 2i + j$, where:
  - $j = (0)/1$ turns (OFF)/ON the verbose mode.
  - $k = (0)/1$ turns (OFF)/ON printing log files. In the default mode, two files `dp.log` and `dp.dat` are generated in the current directory. The file `dp.log` describes the operation of *DP*; upon termination, *DP* writes at the end of `dp.log` the constraints defining the supervisor, and comments on the significance of the supervisor. The file `dp.dat` describes the intermediary Petri nets generated during the operation of *DP*.
  - $l = 0/(1)$ selects between deadlock prevention (default) and liveness or T-liveness enforcement. When $l = 1$, liveness enforcement is performed when the input argument `T` is the empty vector. For T-liveness enforcement, set $l = 1$ and the argument `T` to the set T.
- `how` is either of the strings `'ok'`, `'failed'` or `'impossible'`.
  - The string `'ok'` corresponds to a successful operation of *DP*, in the sense that deadlock is prevented, or that the supervisor is guaranteed to enforce at least T_x-liveness (where the set T_x is reported in `dp.log`, if different from T, in the case of T-liveness enforcement, or if not equal to the total set of transitions, in the case of liveness enforcement).
  - The string `'failed'` corresponds to *DP* failing to generate a supervisor. When this happens, it is possible that no solution exists.
  - The string `'impossible'` corresponds to *DP* detecting that deadlock prevention is impossible. (Note that liveness and T-liveness enforcement imply deadlock prevention.)

Algorithm: *DP* relies on an iterative procedure. At every iteration a class of *siphons* of the net are computed and *controlled*. The procedure is not guaranteed to terminate. The procedure employed in *DP* is a combination of the procedures presented in [3, 4, 1, 5].

M.V. Iordache, P.J. Antsaklis, “Software Tools for the Supervisory Control of Petri Nets Based on Place Invariants,” ISIS Technical Report ISIS-2002-003, April 2002.

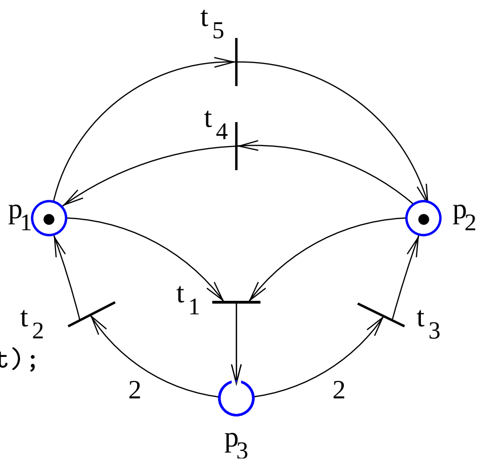
Examples: The Petri net below has controllable transitions and an unobservable transition: t_1. The following code can be used to run the deadlock prevention procedure and to run the liveness enforcement procedure:

```
D = [-1 1 0; -1 0 1; 2 -1 -1];
[Dm, Dp] = d2dd(D);
Tuc = []; Tuo = 1;
pn = getpn(Dm, Dp, Tuc, Tuo);

% Run the deadlock prevention procedure

[L, b, L0, b0, how] = dp(pn);

% Run the liveness enforcement procedure

opt = 15;
[L, b, L0, b0, how] = dp(pn, [], [], [], [], [], opt);
```



Let $T = \{t_4, t_5\}$. To enforce T-liveness in the Petri net at the right, the following code can be used:

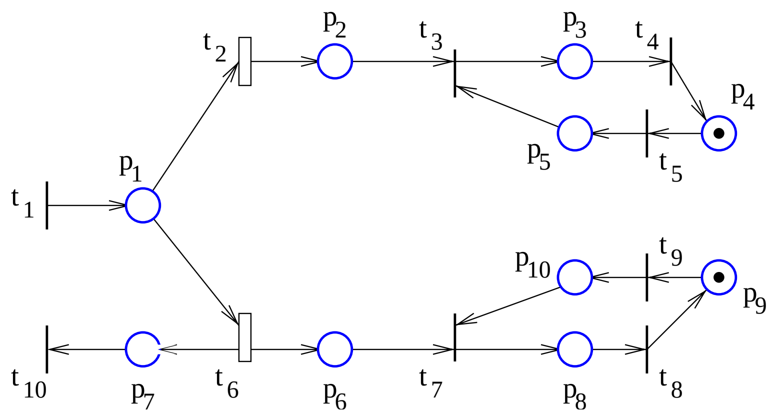
```
D = [-1 1 0 1 -1; -1 0 1 -1 1; 1 -2 -2 0 0];
T = [4, 5];
pn = getpn(D);
opt = 15;
[L, b, L0, b0, how] = dp(pn, T, [], [], [], [], opt);
```



The next example is from [8], pp. 122-129. The following code shows how to obtain the solution using LINENF and DP. The problem is as follows. The following Petri net has observable transitions and two uncontrollable transitions: t_2 and t_6. It is desired to enforce the constraints:

$$\begin{aligned} \mu_5 + \mu_{10} &\leq 1 \\ \mu_2 + \mu_6 &\leq 1 \\ \mu_1 + \mu_7 &\leq 1 \end{aligned}$$

The constraints are inadmissible, and so are transormed to admissible constraints using LINENF. As the closed-loop is blocking, DP is used to add new constraints such that the closed-loop is deadlock-free. The code follows.



```
D = [ 1 -1  0  0  0 -1  0  0  0  0;
      0  1 -1  0  0  0  0  0  0  0;
      0  0  1 -1  0  0  0  0  0  0;
      0  0  0  1 -1  0  0  0  0  0;
      0  0 -1  0  1  0  0  0  0  0;

      0  0  0  0  0  1 -1  0  0  0;
      0  0  0  0  0  1  0  0  0 -1;
      0  0  0  0  0  0  1 -1  0  0;
      0  0  0  0  0  0  0  1 -1  0;
      0  0  0  0  0  0 -1  0  1  0;];

[Dm, Dp] = d2dd(D); Tuc = [2,6];
pn = getpn(Dm, Dp, Tuc);

L = [0 0 0 0 1 0 0 0 0 1;
     0 1 0 0 0 1 0 0 0 0;
     1 0 0 0 0 0 1 0 0 0;];
b = [1 1 1]';

% Obtain admissible constraints Lf*m <= bf

[Dmf,Dpf,ms0,how,dhow,Lf,Cf,bf] = linenf(pn, L, b);

% Apply the deadlock prevention procedure

[L, b, L0, b0, how] = dp(pn, [], -Lf, -bf);
```

Demonstration Scripts: `DPDEM` and `LEDEM` are simple demonstration scripts for deadlock prevention and (T-)liveness enforcement, respectively.

Complete List of Functions and Scripts

ACTN - Computation of a maximal active subnet.

ADMCON - Transformation to admissible constraints.

AR2PN - Conversion from net representation to incidence matrix representation.

ASIPH - Computation of minimal or minimal active siphons of a Petri net.

AVPR - Converts vectors to strings.

CHK_CONS - Checks constraint consistency.

CHK_CON2 - Checks constraint consistency.

CHK_DATA - Checks data consistency.

D2DD - Transforms an incidence matrix D into a input and output matrix pair (D^-, D^+).

DISGRAPH - Text display of coverability graph objects.

DP - Generates a supervisor for deadlock prevention, liveness enforcement, or T-liveness enforcement.

DPDEM - Demonstration program for DP.

FVPR - Coverts vectors to strings.

GCDV - Greatest common divisor of the elements of a vector/matrix.

GETPN - Creates a Petri net object.

GRP - Checks a pattern occurrence in a string.

ILP_ADM - Transformation to admissible constraints based on integer programming.

INVAR - Computes a basis of positive invariants of a Petri net.

IP_SOLVE - Mixed integer program solver based on LP_SOLVE.

ISADM - Checks the admissibility of a set of constraints.

ISADMTST - Demonstration program for ISADM.

LEDEM - Demonstration program for DP.

LINENF - Enforces linear constraints on a Petri net.

LINTST - Demonstration program for LINENF.

MRO_ADM - Transformation to admissible constraints based on matrix row operations.

MSPLIT - Transformation to partially ordinary nets.

NLTRANS - Computes transitions which cannot be made live.

PN2ACPN - Transformation to asymmetric-choice nets.

PN2AR - Conversion from incidence matrix representation to net representation.

PNCGRAPH - Computes the coverability/reachability graph.

PNPRED - Subroutine of PNCGRAPH.

REDUCE - Removes redundant constraints.

SUPERVIS - Enforces linear constraints in fully controllable and observable Petri nets.

TACTN - Computation of a T*-minimal active subnet.*

TS_ADM - Subroutine of ISADM.

References

[1] M. V. Iordache and P. J. Antsaklis. $\mathcal{T}$-liveness enforcement in Petri nets based on structural net properties. In *Proceedings of the 40th IEEE International Conference on Decision and Control.*, December 2001.

[2] M. V. Iordache and P. J. Antsaklis. Synthesis of supervisors enforcing general linear vector constraints in Petri nets. In *Proceedings of the 2002 American Control Conference*, pages 154–159, May 2002.

[3] M. V. Iordache, J. O. Moody, and P. J. Antsaklis. A method for the synthesis of deadlock prevention controllers in systems modeled by Petri nets. In *Proceedings of the 2000 American Control Conference*, pages 3167–3171, June 2000.

[4] M. V. Iordache, J. O. Moody, and P. J. Antsaklis. A method for the synthesis of liveness enforcing supervisors in Petri nets. In *Proceedings of the 2001 American Control Conference*, pages 4943–4948, June 2001.

[5] M. V. Iordache, J. O. Moody, and P. J. Antsaklis. Synthesis of deadlock prevention supervisors using Petri nets. *IEEE Transactions on Robotics and Automation*, 18(1):59–68, February 2002.

[6] M. Jantzen and R. Valk. Formal properties of place/transition nets. In Brauer, W., editor, *Lecture Notes in Computer Science: Net Theory and Applications, Proc. of the Advanced Course on General Net Theory of Processes and Systems, Hamburg, 1979*, volume 84, pages 165–212, Berlin, Heidelberg, New York, 1980. Springer-Verlag.

[7] J. Martinez and M. Silva Suarez. A simple and fast algorithm to obtain all invariants of a generalized petri net. In Girault, C. and Reisig, W., editors, *Informatik-Fachberichte 52: Application and Theory of Petri Nets: Selected Papers from the First and Second European Workshop on Application and Theory of Petri Nets, Strasbourg, Sep. 23-26, 1980, Bad Honnef, Sep. 28-30, 1981*, pages 301–310. Springer-Verlag, 1982.

[8] J. O. Moody and P. J. Antsaklis. *Supervisory Control of Discrete Event Systems Using Petri Nets*. Kluwer Academic Publishers, 1998.

[9] W. Reisig. *Petri Nets*, volume 4 of *EATCS Monographs on Theoretical Computer Science*. Springer-Verlag, 1985.

[10] E. Yamalidou, J. O. Moody, P. J. Antsaklis, and M. D. Lemmon. Feedback control of Petri nets based on place invariants. *Automatica*, 32(1):15–28, January 1996.